Davikky_Teleportation

October 12, 2022

1 Quantum Teleportation

1.1 Contents

[1]:
```
# Do some imports
import numpy as np
import matplotlib
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import IBMQ, Aer, transpile, assemble
from qiskit.visualization import plot_histogram, plot_bloch_multivector,␣
 ↪array_to_latex
from qiskit.extensions import Initialize
from qiskit.quantum_info import random_statevector
```

1.2 1. Overview

Quantum teleportation is a technique for transmitting quantum information from a sender at one location to a receiver, whereby the distance is at least theoretically irrelevant. In contrast to the general understanding, no physical objects are transported from one place to the next, but only quantum information is transmitted.

The sender does not need to know the particular quantum state being transmitted. Furthermore, the location of the receiver can be unknown. However, since classical information must also be sent from the sender to the receiver in order to complete the teleportation, teleportation cannot take place faster than the speed of light.

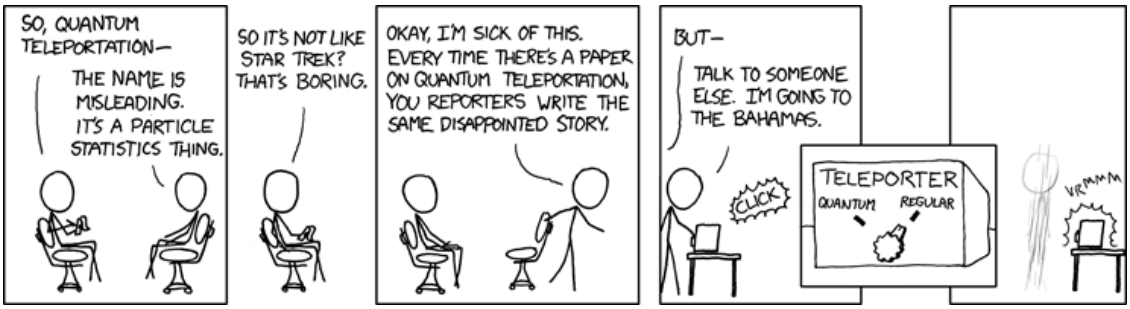
Bell states and their entanglement form the basis of quantum teleportation.

1.2.1 1.1 Quantum Teleportation in general

Quantum teleportation is a concept that takes advantage of the powers of quantum mechanics. It is the key behind quantum key distribution and long distance quantum communication, but also

used in quantum computing. It allows you to **move** a quantum state from one qubit to another.

Why do we need this procedure?
In classical computing, we can copy the information from one bit to another. In quantum computing, this is impossible: If we read out a quantum state, the superposition and entanglement is destroyed. In addition, the **no-cloning theorem** states that it is not possible to copy an unknown quantum state exactly. Quantum teleportation is actually **not** teleporting or copying information. It actually means, we move the quantum state from one qubit to another - while destroying the state of the first qubit, so the no-cloning theorem is not violated.



Source: https://www.explainxkcd.com/

1.2.2 1.2 Quantum Teleportation Protocol

Let us assume, Alice wants to quantum teleport some information to Bob, who is some distance away.
It can be any unknown state, i.e. we can apply any sequence of x,y,z, Hadamard, phase etc gates to get any arbitrary state.

Assume, Alice wants to send the qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, i.e. passing on information about α and β to Bob.

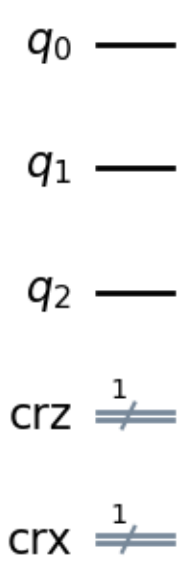
To transfer a quantum bit, Alice and Bob must use an entangled qubit pair. Alice then performs some operations on her qubit, sends the results to Bob over a classical communication channel, and Bob then performs some operations on his end to receive Alice's qubit.

Requirements: In total we need 3 qubits and 2 classical bits. Two of the qubits must be entangled, i.e. the qubit that Alice has and the qubit that Bob has.

Recap Entanglement: The statement of entangled particles basically means, that the states of the two are *linked* or in other words they are not separable. This means that the state of one qubit can't be described without the state of the other. Intuition: Information is spread across this particles.

[2]:
```
## SETUP
# Protocol uses 3 qubits and 2 classical bits in 2 different registers

qr = QuantumRegister(3, name="q")    # Protocol uses 3 qubits
crz = ClassicalRegister(1, name="crz") # and 2 classical bits
crx = ClassicalRegister(1, name="crx") # in 2 different registers
teleportation_circuit = QuantumCircuit(qr, crz, crx)
```

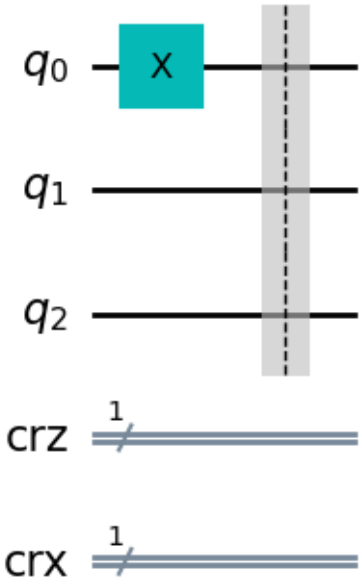
[3]:
```
teleportation_circuit.draw(output='mpl')
```
[3]:



q_0: Message Alice wants to send
q_1: The qubit Alice owns
q_2: Bob's qubit

Step 0 We apply some sequence on the message Qubit.

[4]:
```
teleportation_circuit.x(0) ## apply some transformation to the qubit message
teleportation_circuit.barrier()
teleportation_circuit.draw(output='mpl')
```
[4]:

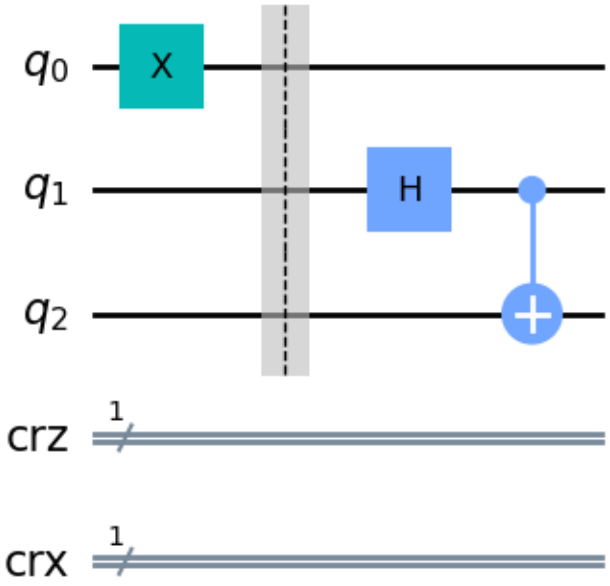


Message Qubit is now in state 1.

Step 1 We want to teleport this information to Bob. We need to entangle Alice's qubit and Bobs qubit, i.e. q_1 and q_2 - this results in a **Bell pair**. Bob's qubit is **physically** transported to Bob.

[5]:
```
# create bell pair for a and b in circuit qc
def create_bell_pair(qc, a, b):
    qc.h(a) # Put qubit a into state |+>
    qc.cx(a,b) # CNOT with a as control and b as target

create_bell_pair(teleportation_circuit, 1, 2)
teleportation_circuit.draw(output='mpl')
```

[5]:



Assume, Alice owns q_1 and Bob owns q_2 after they part ways.

Step 2 - Bell Measurement Alice owns the original message q_0 and shares its entangled qubit q_1, one half of the Bell pair, i.e. of the entangled qubits.

Now we perform a **Bell measurement** on the two qubits, that Alice has. Bell Measurement: Quantum mechanical measurement, that determines in which of the four Bell states the two qubits are in.

Alice applies a CNOT gate to q_1, controlled by q_0 (the qubit she is trying to send Bob), and a Hadamard gate to q_0:

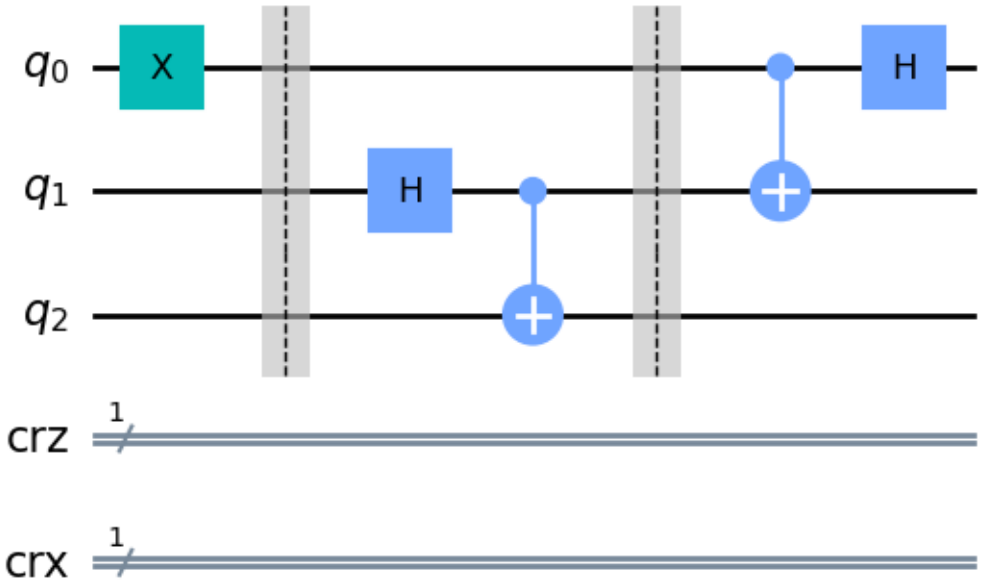
[6]:
```
def alice_gates(qc, psi, a):
    qc.cx(psi, a)
    qc.h(psi)
```

This CNOT unentangles the two qubits. Quantum information is translated back to classical information.

[7]:
```
## SETUP
# Protocol uses 3 qubits and 2 classical bits in 2 different registers
qr = QuantumRegister(3, name="q")
crz, crx = ClassicalRegister(1, name="crz"), ClassicalRegister(1, name="crx")
teleportation_circuit = QuantumCircuit(qr, crz, crx)

## STEP 0
```

```
teleportation_circuit.x(0) ## apply some transformation to the qubit message
teleportation_circuit.barrier()
## STEP 1
create_bell_pair(teleportation_circuit, 1, 2)

## STEP 2
teleportation_circuit.barrier() # Use barrier to separate steps
alice_gates(teleportation_circuit, 0, 1)
teleportation_circuit.draw(output='mpl')
```

[7]:



Step 3 When Alice measures her qubits (q_1 and q_0), there's going to be 4 possible combinations: 00, 01, 10, 11 - the Bell states.

Alice stores her measurement in two classical bits. She then sends these two bits to Bob: It **CANNOT** be faster than the speed of light.

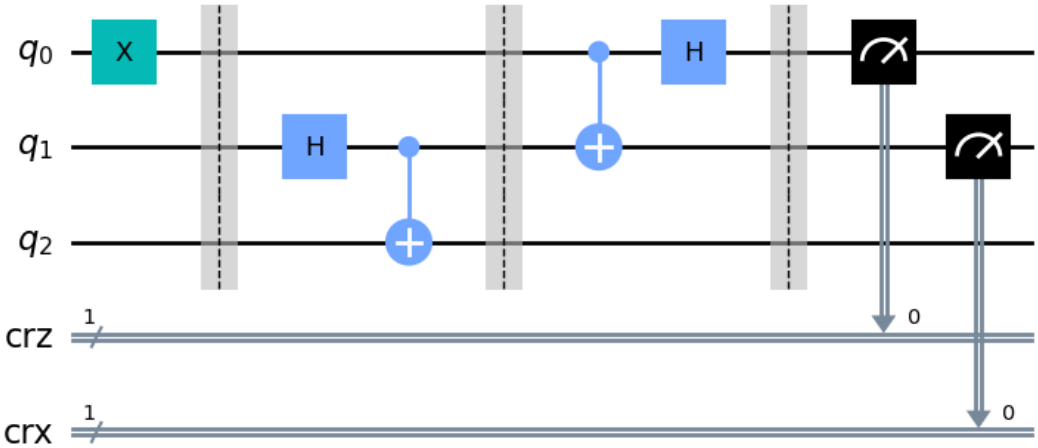
The result from the message is the first bit, and the result is the second bit.

[8]:
```
def measure_and_send(qc, a, b):
    """Measures qubits a & b and 'sends' the results to Bob"""
    qc.measure(a,0) #measure qubit a and store in first cbit, crz
    qc.measure(b,1) #measure qubit b and store in sencond cbit, crx
```

[9]:
```
## SETUP
# Protocol uses 3 qubits and 2 classical bits in 2 different registers
```

```
qr = QuantumRegister(3, name="q")
crz, crx = ClassicalRegister(1, name="crz"), ClassicalRegister(1, name="crx")
teleportation_circuit = QuantumCircuit(qr, crz, crx)

## STEP 0
teleportation_circuit.x(0) ## apply some transformation to the qubit message
teleportation_circuit.barrier()
## STEP 1
create_bell_pair(teleportation_circuit, 1, 2)

## STEP 2
teleportation_circuit.barrier() # Use barrier to separate steps
alice_gates(teleportation_circuit, 0, 1)
teleportation_circuit.barrier()

## STEP 3
measure_and_send(teleportation_circuit, 0 ,1)
teleportation_circuit.draw(output='mpl')
```

[9]:



Step 4 - Recreate the Quantum State When you measure 2 entangled stated, they collapse. We don't know how to interpret the measurement and the correlations until we get the classical information from Alice.

After the measurement, depending on what Bob received, he does some operations to his qubit. Bob, who already has the qubit q_2, then applies the following gates depending on the state of the classical bits:

00 $\rightarrow$ Do nothing

01 $\rightarrow$ Apply X gate

10 → Apply Z gate

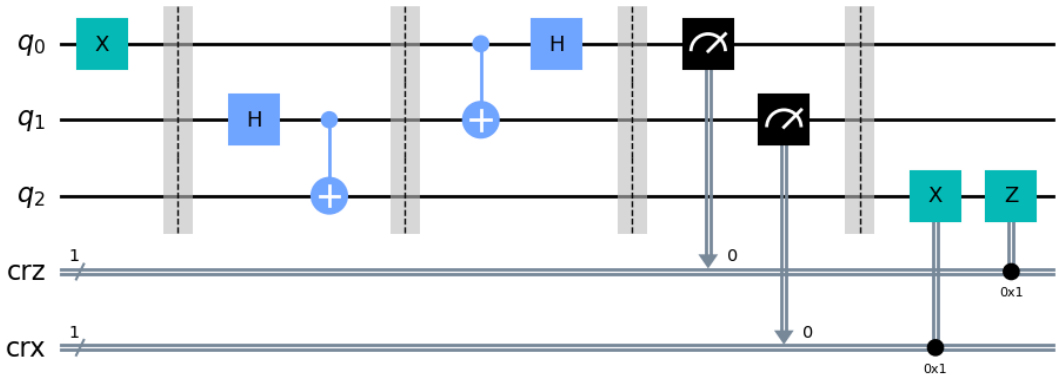
11 → Apply ZX gate

cbits are read 'crz-crx' in that order

[10]:
```
# This function takes a QuantumCircuit (qc), integer (qubit)
# and ClassicalRegisters (crz & crx) to decide which gates to apply
def bob_gates(qc, qubit, crz, crx):
    # Here we use c_if to control our gates with a classical
    # bit instead of a qubit
    qc.x(qubit).c_if(crx, 1) # Apply gates if the registers
    qc.z(qubit).c_if(crz, 1) # are in the state '1' , cbits are read 'crz-crx'␣
 ↪in that order
```

[11]:
```
## SETUP
# Protocol uses 3 qubits and 2 classical bits in 2 different registers
qr = QuantumRegister(3, name="q")
crz, crx = ClassicalRegister(1, name="crz"), ClassicalRegister(1, name="crx")
teleportation_circuit = QuantumCircuit(qr, crz, crx)

## STEP 0
teleportation_circuit.x(0) ## apply some transformation to the qubit message

teleportation_circuit.barrier()
## STEP 1
create_bell_pair(teleportation_circuit, 1, 2)

teleportation_circuit.barrier() # Use barrier to separate steps
## STEP 2
alice_gates(teleportation_circuit, 0, 1)

teleportation_circuit.barrier() # Use barrier to separate steps
## STEP 3
measure_and_send(teleportation_circuit, 0, 1)

teleportation_circuit.barrier() # Use barrier to separate steps
## STEP 4
bob_gates(teleportation_circuit, 2, crz, crx)

teleportation_circuit.draw(output='mpl')
```

[11]:



Now, Bob has a qubit with the original state that Alice's qubit had.

And voila! At the end of this protocol, Alice's **unknown message** has now teleported to Bob.

Why does it work? Alice's and Bob's state depend on each other. The classical information Alice receives from Bob tells how his state differs from Alice's. That tells him, which gates he needs to apply to get back to the original state.

Key concept: Bell pairs. Gives information about how measurements are correlated. The correlation between these pairs means that the results will always be equal. If one of the qubits of the Bell pair is 0, the other one is 0, if one is 1 the other one is 1 and vice versa.

Maybe confusing: They have to be in the same basis. Basis states: components of how we can measure any state in the system using these 2 components, i.e. any combination of these two to get to any point in the vector space. - Alice classical information send tells us how we need to change our basis state to get back to the situation where Alice's and Bob's qubit are perfectly correlated. - we need to make sure that Bob gets back to the right basis state where they're perfectly correlated - Message qubit and Bob's qubit are gonna be the same - Message qubit is collapsed and his unknown state is transferred to Bob - no-cloning theorem is not violated

1.3 2. Simulating the Teleportation Protocol

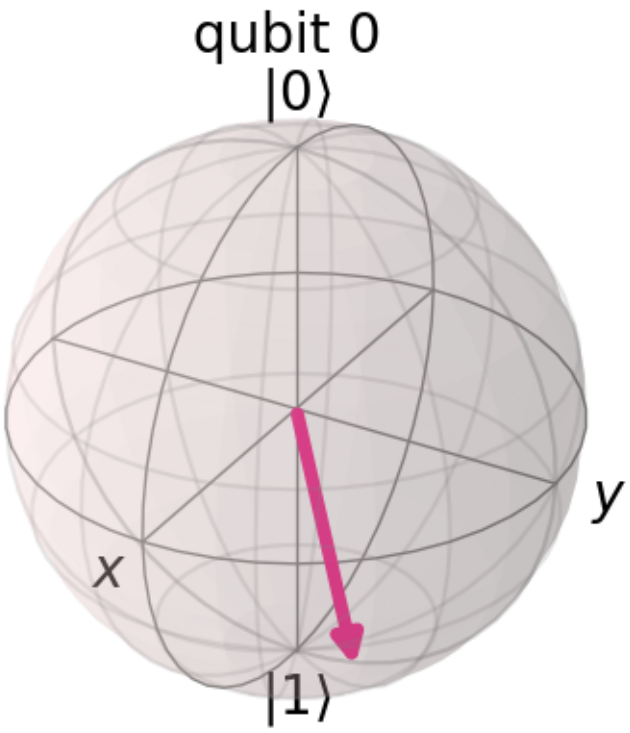
1.3.1 2.1 How Will We Test the Protocol on a Quantum Computer?

We initialize Alice's message qubit in a random state $|\psi\rangle$ (`psi`). Remark: it can be any unknown state.

[12]:
```
# Create random 1-qubit state
psi = random_statevector(2)

# Display it nicely
display(array_to_latex(psi, prefix="|\\psi\\rangle ="))
# Show it on a Bloch sphere
plot_bloch_multivector(psi)
```

$$|\psi\rangle = \begin{bmatrix} 0.11083 - 0.09977i & 0.82112 + 0.55093i \end{bmatrix}$$

[12]:



Let's create our initialization instruction to create $|\psi\rangle$ from the state $|0\rangle$:

[13]:
```
init_gate = Initialize(psi) # instruction operation - technically not a gate
init_gate.label = "init"
```

At the end of the circuit the qubit $|q_2\rangle$ will be in this state. We will check this using the statevector simulator.

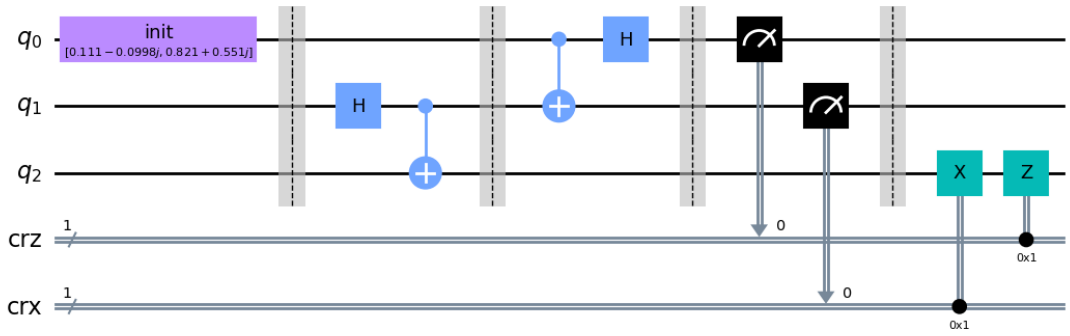
1.3.2 3.2 Using the Simulated Statevector

We can use the Aer simulator to verify our qubit has been teleported.

[14]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
 ↪different name to ensure no overlaps :)
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
```

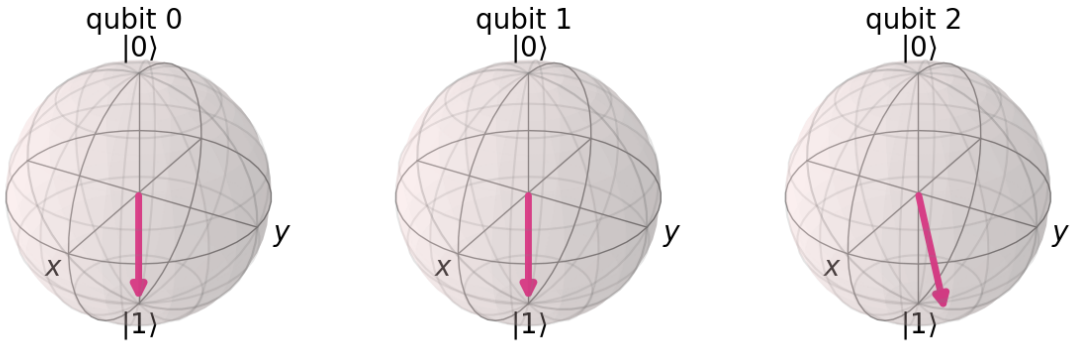
```
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.barrier()

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)
qc.barrier()
## STEP 3
# Alice performs a bell measurement and sends her classical bits to Bob
measure_and_send(qc, 0, 1)
qc.barrier()
## STEP 4
# Bob recreates the quantum state
bob_gates(qc, 2, crz, crx)

# Display the circuit
qc.draw(output='mpl')
```

[14]:



[15]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector)
```

[15]:



Qubit q_0 and q_1 have collapsed to $|0\rangle$ or $|1\rangle$. Bob's Qubit q_2 is now in the same state as the message was before we did anything - compare to 3.1

The state $|\psi\rangle$ has been teleported from qubit 0 to qubit 2.

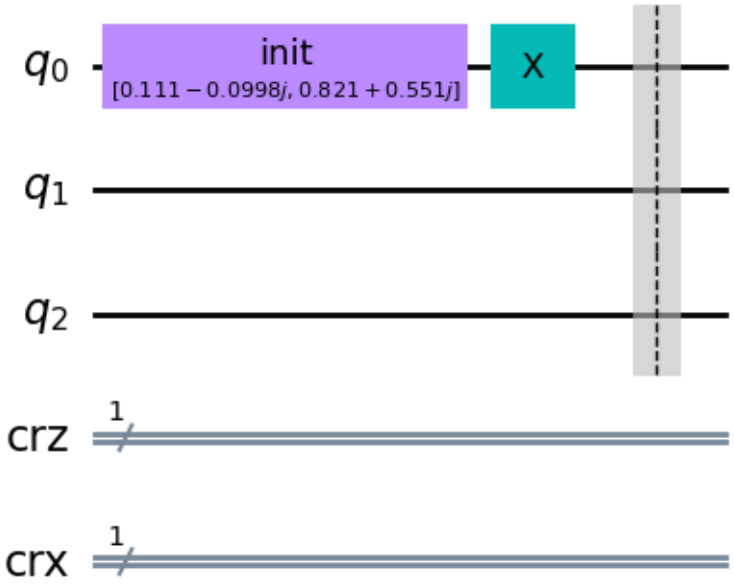
1.3.3 EXERCISE

Apply a series of gates to the message and check whether the information, i.e. the final state is teleported correctly.

1. USING X-GATE, Preview the Circuit diagram for the message before Teleportation

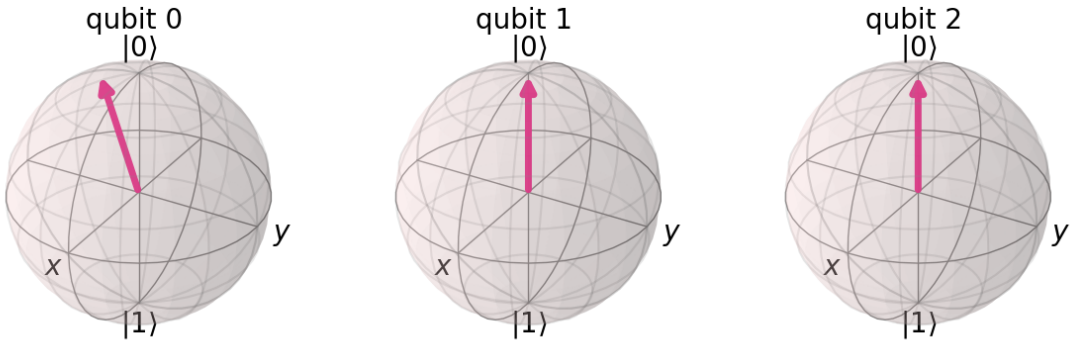
[17]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
 ↪different name to ensure no overlaps :)
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.x(0)
qc.barrier()

qc.draw(output='mpl')
```

[17]:



Now, Preview the state of the message before Teleportation

[18]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector2 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector2)
```

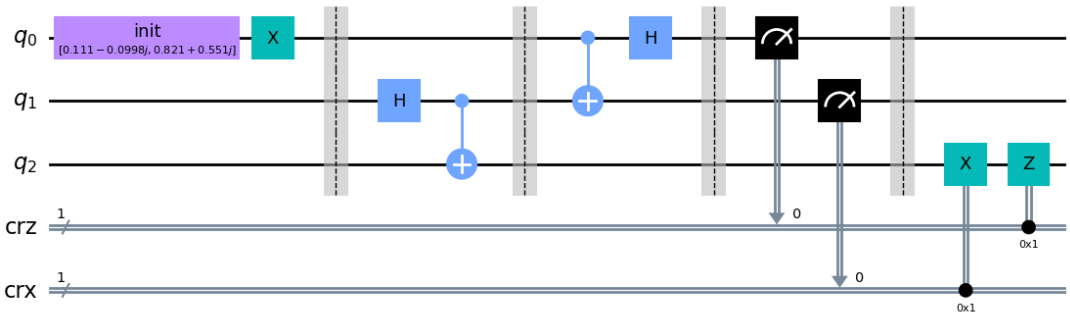
[18]:


Implementing the teleportation protocol...

[22]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
↪different name to ensure no overlaps :)
```

```
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.x(0)
qc.barrier()

qc.draw(output='mpl')

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)
qc.barrier()
## STEP 3
# Alice performs a bell measurement and sends her classical bits to Bob
measure_and_send(qc, 0, 1)
qc.barrier()
## STEP 4
# Bob recreates the quantum state
bob_gates(qc, 2, crz, crx)

# Display the circuit
qc.draw(output='mpl')
```

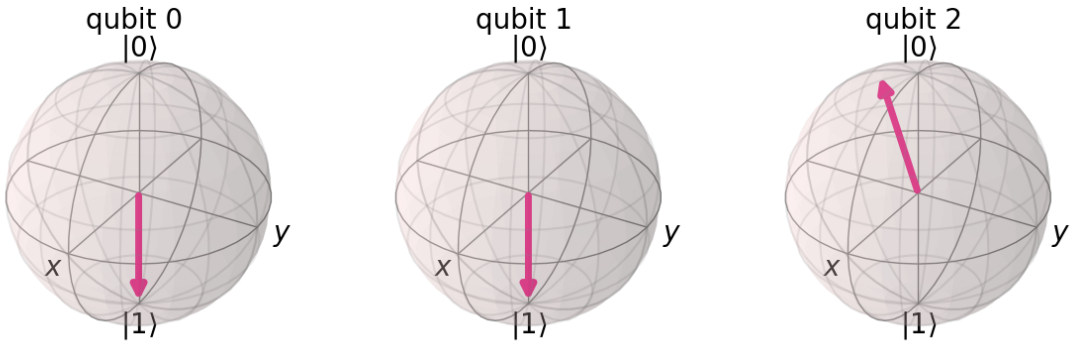
[22]:



Now, Preview the state of Bob's qubit (or the message) after Teleportation

[23]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector2 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector2)
```

[23]:

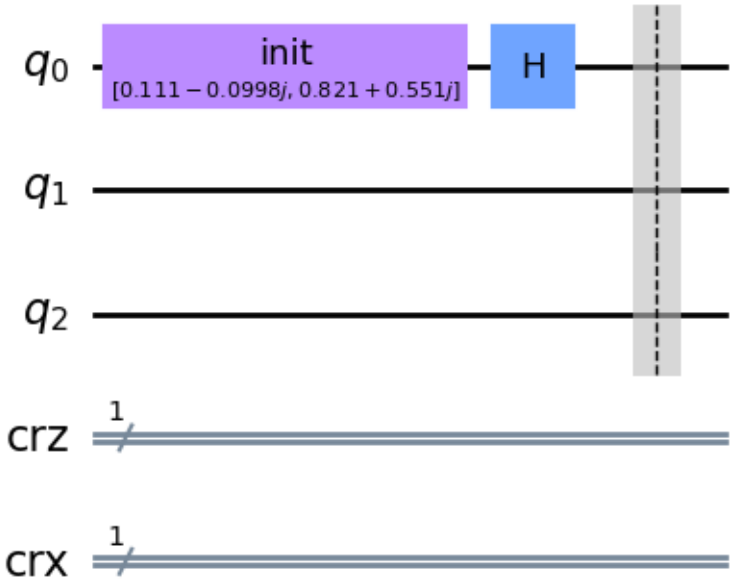


1.3.4

2. USING H-GATE, Preview the Circuit diagram for the message before Teleportation

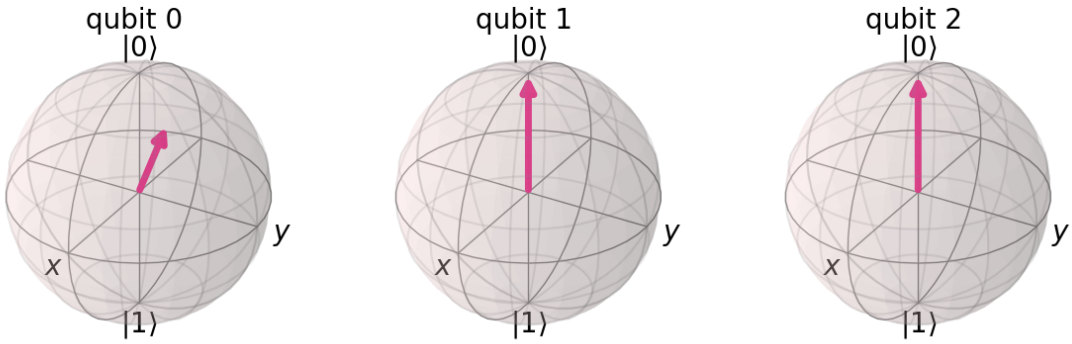
[24]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
 ↪different name to ensure no overlaps :)
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.h(0)
qc.barrier()

qc.draw(output='mpl')
```

[24]:



Now, Preview the state of the message before Teleportation

[25]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector3 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector3)
```

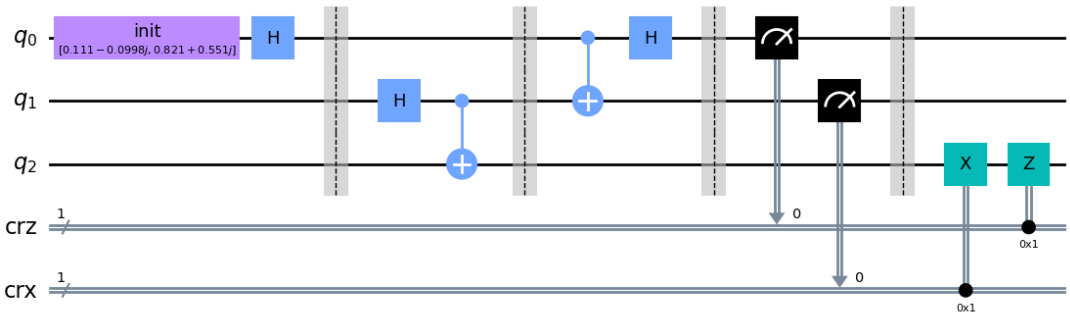
[25]:


Implementing the teleportation protocol...

[26]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
↪different name to ensure no overlaps :)
```

```
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.h(0)
qc.barrier()

qc.draw(output='mpl')

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)
qc.barrier()
## STEP 3
# Alice performs a bell measurement and sends her classical bits to Bob
measure_and_send(qc, 0, 1)
qc.barrier()
## STEP 4
# Bob recreates the quantum state
bob_gates(qc, 2, crz, crx)

# Display the circuit
qc.draw(output='mpl')
```

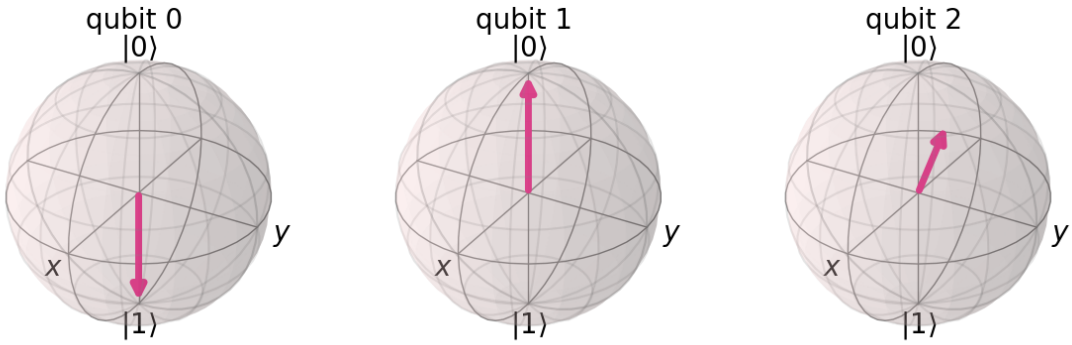
[26]:



Now, Preview the state of Bob's qubit (or the message) after Teleportation

[27]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector3 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector3)
```

[27]:



1.3.5

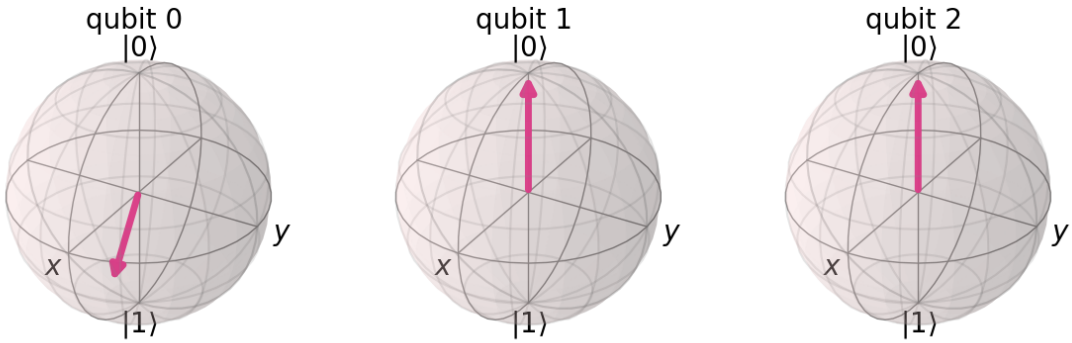
3. USING Z-GATE, Preview the Circuit diagram for the message before Teleportation

[28]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
 ↪different name to ensure no overlaps :)
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.z(0)
qc.barrier()

qc.draw(output='mpl')
```

[28]:

Now, Preview the state of the message before Teleportation

[29]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector4 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector4)
```

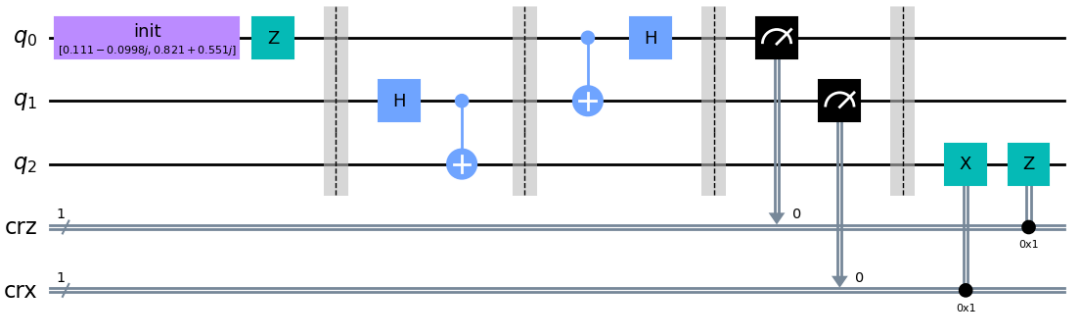
[29]:


Implementing the teleportation protocol...

[30]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
 ↪different name to ensure no overlaps :)
```

```
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.z(0)
qc.barrier()

qc.draw(output='mpl')

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)
qc.barrier()
## STEP 3
# Alice performs a bell measurement and sends her classical bits to Bob
measure_and_send(qc, 0, 1)
qc.barrier()
## STEP 4
# Bob recreates the quantum state
bob_gates(qc, 2, crz, crx)

# Display the circuit
qc.draw(output='mpl')
```

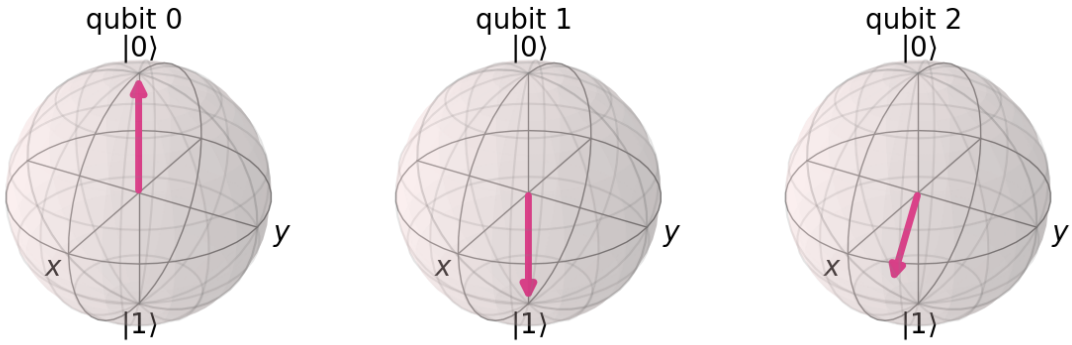
[30]:



Now, Preview the state of Bob's qubit (or the message) after Teleportation

[31]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector4 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector4)
```

[31]:

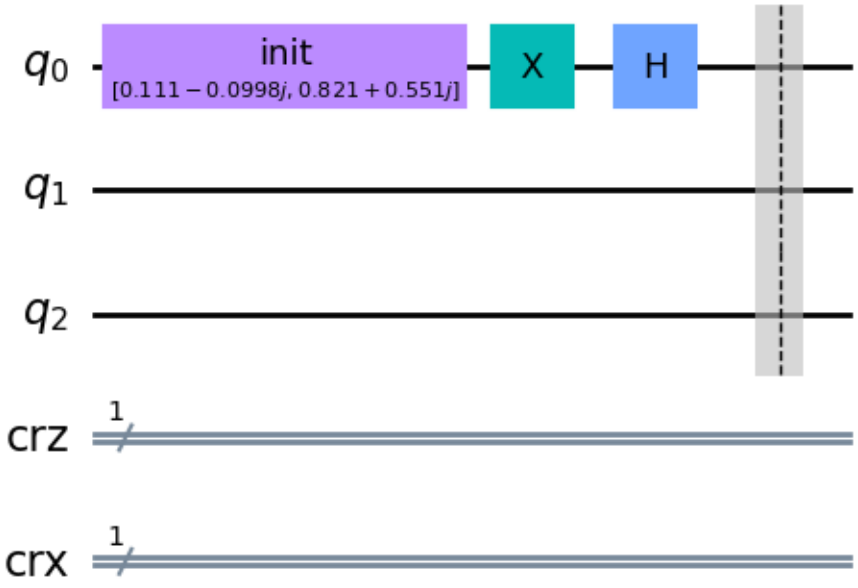


1.3.6

3. USING HX-GATE, Preview the Circuit diagram for the message before Teleportation

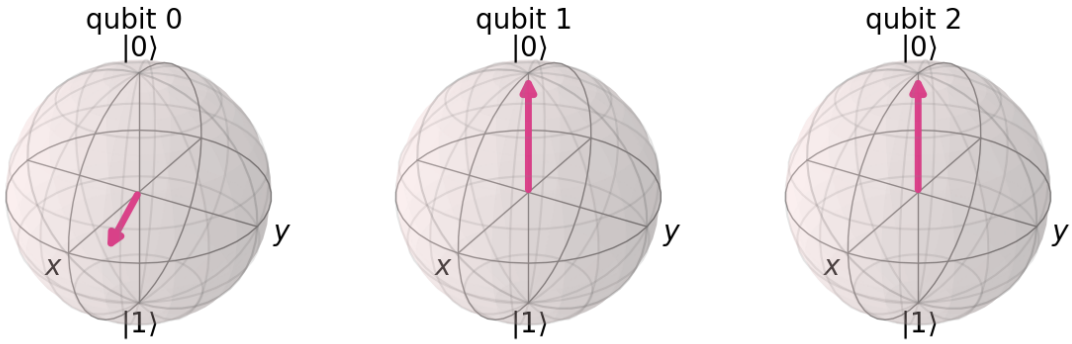
[32]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
 ↪different name to ensure no overlaps :)
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.x(0)
qc.h(0)
qc.barrier()

qc.draw(output='mpl')
```

[32]:



Now, Preview the state of the message before Teleportation

[33]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector5 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector5)
```

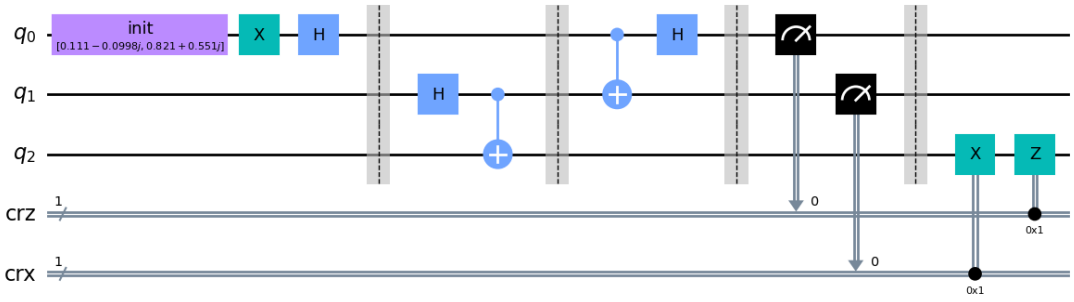
[33]:


Implementing the teleportation protocol...

[34]:
```
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits -give it a␣
 ↪different name to ensure no overlaps :)
```

```
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's message in q0
qc.append(init_gate, [0])
qc.x(0)
qc.h(0)
qc.barrier()

qc.draw(output='mpl')

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)
qc.barrier()
## STEP 3
# Alice performs a bell measurement and sends her classical bits to Bob
measure_and_send(qc, 0, 1)
qc.barrier()
## STEP 4
# Bob recreates the quantum state
bob_gates(qc, 2, crz, crx)

# Display the circuit
qc.draw(output='mpl')
```

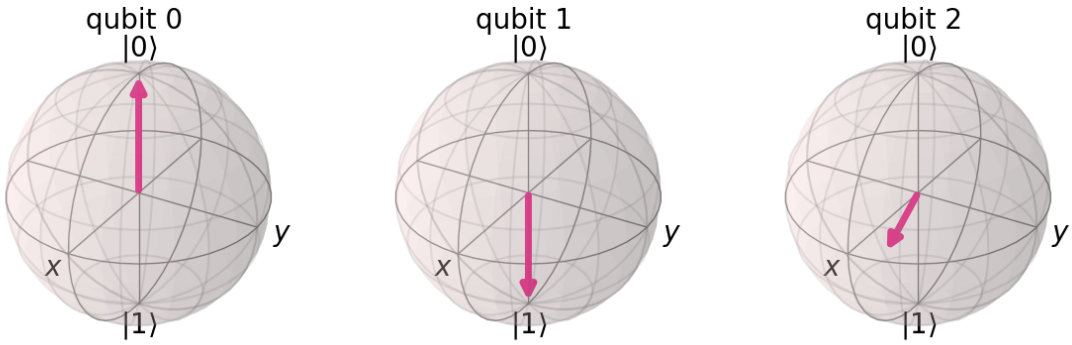
[34]:



Now, Preview the state of Bob's qubit (or the message) after Teleportation

[35]:
```
sim = Aer.get_backend('aer_simulator')
qc.save_statevector()
out_vector5 = sim.run(qc).result().get_statevector()
plot_bloch_multivector(out_vector5)
```

[35]:



[ ]: